Concurrency and Parallelism

- PL support for concurrency/parallelism a huge topic
  - Increasingly important (not traditionally in PL courses)
  - Lots of active research as well as decades-old work
- We’ll just do explicit threads plus:
  - Shared memory (locks and transactions)
  - Futures
  - Synchronous message passing (Concurrent ML)
- We’ll skip
  - Process calculi (foundational message-passing)
  - Asynchronous methods, join calculus, ...
  - Data-parallel languages (e.g., NESL or ZPL)
  - ...
- Mostly in ML syntax (inference rules where convenient)
  - Even though current Caml implementation has threads but not parallelism

Concurrency vs. Parallelism

(Terminology not universal, but distinction paramount):

**Concurrency** is about correctly and efficiently managing access to shared resources
- Examples: operating system, shared hashtable, version control
- Key challenge is responsiveness to external events that may arrive asynchronously and/or simultaneously
- Often provide responsiveness via threads
- Often focus on synchronization

**Parallelism** is about using extra computational resources to do more useful work per unit time
- Examples: scientific computing, most graphics, a lot of servers
- Key challenge is Amdahl’s Law (no sequential bottlenecks)
- Often provide parallelism via threads on different processors and/or to mask I/O latency

Threads

High-level: “Communicating sequential processes”

Low-level: “Multiple stacks plus communication”

From Caml’s thread.mli:

```plaintext
type t (*thread handle; remember we’re in module Thread*)
val create : ('a->'b) -> 'a -> t (* run new thread *)
val self : unit -> t (* what thread is executing this? *)
The code for a thread is in a closure (with hidden fields) and
Thread.create actually spawns the thread
```

Most languages make the same distinction, e.g., Java:

- Create a Thread object (data in fields) with a run method
- Call its start method to actually spawn the thread

One possible formalism (omitting thread-ids)

- Program state is one heap and multiple expressions
- Any ei might “take the next step” and potentially spawn a thread
- A value in the “thread-pool” is removable
- Nondeterministic with interleaving granularity determined by rules

Some example rules for \( H; e \rightarrow H'; e'; o \) (where \( o ::= \cdot \mid e \)):

\[
\begin{align*}
H; e_1 \rightarrow H'; e'_1; o \\
H; e_1 \rightarrow H'; e'_1; o \\
H; spawn(v_1, v_2) \rightarrow H; 0; (v_1 \ v_2)
\end{align*}
\]

Why use threads?

One OR more of:
1. Performance (multiprocessor or mask I/O latency)
2. Isolation (separate errors or responsiveness)
3. Natural code structure (1 stack awkward)

It’s not just performance

On the other hand, it seems fundamentally harder (for programmers, language implementors, language designers, semanticists) to have multiple threads of execution

Examples: scientific computing, most graphics, a lot of servers

- Create a thread object (data in fields) with a run method
- Call its start method to actually spawn the thread

Mostly in ML syntax (inference rules where convenient)
- Even though current Caml implementation has threads but not parallelism
Formalism continued

The $H; e \rightarrow H'; e'$ o judgment is just a helper-judgment for $H; T \rightarrow H'; T'$ where $T ::= \cdot | e; T$

$$
H; e \rightarrow H'; e';
\frac{H; e_1; \ldots ; e_i; \ldots ; e_n \rightarrow H; e_1; \ldots ; e_i'; \ldots ; e_n}{H' ; e_1; \ldots ; e_i; \ldots ; e_n \rightarrow H'; e_1; \ldots ; e_i'; \ldots ; e_n; e''}
\frac{H; e \rightarrow H'; e; e''}{H; e_1; \ldots ; e_i; \ldots ; e_n \rightarrow H; e_1; \ldots ; e_i; \ldots ; e_n; e}\n$$

Program termination: $H; \cdot$

Communication

If threads do nothing other threads need to “see,” we are done

Best to do as little communication as possible

E.g., do not mutate shared data unnecessarily, or hide mutation behind easier-to-use interfaces

One way to communicate: Shared memory

One thread writes to a ref, another reads it

Sounds nasty with pre-emptive scheduling

Hence synchronization mechanisms

Taught in O/S for historical reasons!

Fundamentally about restricting interleavings

Example: Rewriting of Adjacent Data

From “Threads cannot be implemented as a library”
Consider a C “struct” containing bit fields on a little-endian 32-bit machine.

struct {int a:17; int b:15} x;

Because very few (no?) machines support a 17-bit-wide store instruction, the assignment $x.a = 42$ is likely to be implemented by something like this:

```c
{
    tmp = x; /* Read both fields into 32-bit variable */
    tmp &= ~0x1ffff; /* Mask off old a */
    tmp |= 42; /* Overwrite all of x */
    x = tmp;
}
```

Communication

If threads do nothing other threads need to “see,” we are done

Best to do as little communication as possible

E.g., do not mutate shared data unnecessarily, or hide mutation behind easier-to-use interfaces

One way to communicate: Shared memory

One thread writes to a ref, another reads it

Sounds nasty with pre-emptive scheduling

Hence synchronization mechanisms

Taught in O/S for historical reasons!

Fundamentally about restricting interleavings

Example: Rewriting of Adjacent Data (cont.)

Expressions equivalent in a single-threaded world are not necessarily equivalent in a multithreaded context!

Example in Caml:

```caml
{
    tmp = x; /* Read both fields into 32-bit variable */
    tmp &= ~0x1ffff; /* Mask off old a */
    tmp |= 42; /* Overwrite all of x */
    
    x = tmp; /* Overwrite all of x */
}
```

Fine for sequential codes. But what if there is a concurrent update of $x.b$ that occurs between the $tmp=x$ and $x=tmp$ assignments in the above code?

We have introduced a race, and the concurrent update to $x.b$ may be lost, in spite of the fact that the two threads operate on completely distinct fields.

Join

“Fork-join” parallelism a simple approach good for “farm out subcomputations then merge results”

(* suspend caller until/unless arg terminates *)

val join : t -> unit

Common pattern:

```caml
val fork_join : ('a -> 'b array) -> (* divider *)
    ('b -> 'c) -> (* conqueror *)
    ('c array -> 'd) -> (* merger *)
    'a -> (* data *)
    'd
```

Apply the second argument to each element of the 'b array in parallel, then use third argument after they are done.

See lec16code.ml for implementation and related patterns (untested)
Futures

A different model for explicit parallelism without explicit shared memory or message sends

▶ Easy to implement on top of either, but most models are easily inter-implementable
▶ See ML file for implementation over shared memory

```
type 'a promise;
val future : (unit -> 'a) -> 'a promise (*do in parallel*)
val force : 'a promise -> 'a (*may block*)
```

Essentially fork/join with a value returned?

▶ Returning a value more functional
▶ Less structured than "cobegin s1; s2; ... sn" form of fork/join

Getting it wrong

Races can result from too little synchronization

▶ Data races: simultaneous read-write or write-write of same memory location
  ▶ Lots of PL work in last 15 years on types and tools to prevent/detect
  ▶ Provided language has some guarantees, may not be a bug
    ▶ Canonical example: parallel search and "done" bits
    ▶ But few language have such guarantees (!)
▶ Higher-level races: much tougher to prevent in the language
  ▶ Amount of correct nondeterminism inherently app-specific

Deadlock can result from too much synchronization

▶ Cycle of threads waiting for someone else to do something
▶ Easy to detect dynamically with locks, but then what?

Atomic Blocks (Software Transactions)

Java-like: atomic { s }

Caml-like: atomic : (unit -> 'a) -> 'a

Execute the body/thunk as though no interleaving from other threads

▶ Allow parallelism unless there are actual run-time memory conflicts (detect and abort/retry)
▶ Convenience of coarse-grained locking with parallelism of fine-grained locking (or better)
▶ But language implementation has to do more to detect conflicts (much like garbage collection is convenient but has costs)

Most research on implementation (preserve parallelism unless there are conflicts), but this is not an implementation course
Transactions make things easier

Problems like append and xfer become trivial

So mixing coarse-grained and fine-grained operations (e.g., hashtable lookup and hashtable resize)

Transactions are great, but not a panacea:
- Application-level races can remain
- Application-level deadlock can remain
- Implementations generally try-and-abort, which is hard for “launch missiles” (e.g., I/O)
- Many software implementations provide a weaker and under-specified semantics if there are data races with non-transactions
- Memory-consistency model questions remain and may be worse than with locks...

Relaxed memory models

Modern imperative and OO languages do not promise sequential consistency (if they say anything at all)
- The hardware makes it prohibitively expensive
- Renders unsound almost every compiler optimization
  Example: common-subexpression elimination
    Initially a=0 and b=0
    Thread 1  Thread 2
    x=a+b;  b=1;
y=a;  a=1;
z=a+b;
assert(z>=y);

In real languages
- Java: If every sequentially consistent execution of program $P$ is data-race free, then every execution of program $P$ is equivalent to some sequentially consistent execution
  - Not the definition, a theorem about the definition
  - Actual definition very complicated, balancing needs of code writers, compiler optimizers, and hardware
    - Complicated by constructors and final fields
    - Not defined in terms of “list of acceptable optimizations”

- C++: Roughly, any data race is as undefined as an array-bounds error. No such thing as a benign data race and no guarantees if you have one. (In practice, programmers will still assume things, like they do with casts.)
  - But same theorem as Java: “DRF ⇒ SC”

- Most languages: Eerily silent
  - Arguably the greatest current failure of programming languages

Memory models

A memory-consistency model (or just memory model) for a concurrent shared-memory language specifies “which write a read can see”

The gold standard is sequential consistency (Lamport): “the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program”

Under sequential consistency, this assert cannot fail, despite data races:

```
let x, y = ref 0, ref 0
let _ = create (fun () -> x := 1; y := 1) ()
let _ = create (fun () -> let r = !y in let s = !x in assert(s>=r) ()
```

Relaxed ≠ Nothing

But (especially in a safe language) have to promise something
- When is code “correctly synchronized”?
- What can a compiler do in the presence of races?
  - Cannot seg-fault Java or compromise the SecurityManager
  - Can a race between x:=1 and !x cause the latter to produce a value “out of thin air”? (Java: no)

The definitions are very complicated and programmers can usually ignore them, but do not assume sequential consistency

See also Java’s volatiles and C++’s atomics

Mostly functional wins again

If most of your data is immutable and most code is known to access only immutable data, then most code can be optimized without any concern for the memory model

So can afford to be very conservative for the rest

Example: A Caml program that uses mutable memory only for shared-memory communication

Non-example: Java, which uses mutable memory for almost everything
- Compilers try to figure out what is thread-local (again avoids memory-model issues), but it’s not easy
Initially x==0 and y==0
Thread 1     Thread 2
x=1;         r=y;
y=1;         s=x;

Can s be less than r?

Yes

In Java, no
- Notion of “happens-before” ordering between release and acquire of the same lock

Nobody really knows, but often yes (!) in prototype implementations